

Herencia

lección4

La herencia es una forma práctica de reutilizar el código en el mundo de la orientación a objetos. Usar herencia con JPA y OpenXava es tan fácil como hacerlo con puro Java. Vamos a usar la herencia para quitar el código repetitivo y aburrido, como la definición de los UUID. También, añadiremos una nueva entidad, `Order`, y usaremos la herencia para hacerlo con muy poco código. Además, aprenderás cuán práctico es usar herencia incluso para código de pruebas.

4.1 Heredar de una superclase mapeada

Las clases `Author`, `Category`, `Detail` e `Invoice` tienen algo de código en común. Este código es la definición del campo `oid` (listado 4.1) y es exactamente el mismo para todas estas clases. Ya sabes que copiar y pegar es un pecado mortal, por eso tenemos que buscar una forma de quitar este código repetido, y así evitar ir al infierno.

Listado 4.1 Propiedad oid: código común para `Author`, `Category`, `Detail` e `Invoice`

```
@Id @GeneratedValue(generator="system-uuid") @Hidden
@GenericGenerator(name="system-uuid", strategy = "uuid")
@Column(length=32)
private String oid;

public String getOid() {
    return oid;
}

public void setOid(String oid) {
    this.oid = oid;
}
```

Una solución elegante en este caso es usar herencia. JPA permite varias formas de herencia. Una de ellas es heredar de una superclase mapeada. Una superclase mapeada es una clase Java con anotaciones de mapeo JPA, pero no es una entidad en sí. Su único objetivo es ser usada como clase base para definir entidades. Usemos una, y verás su utilidad rápidamente.

Primero, movemos el código común a una clase marcada como `@MappedSuperclass`. La llamamos `Identifiable`⁶. La puedes ver en listado 4.2.

Listado 4.2 `Identifiable`: una superclase mapeada con generación de UUID

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.*;
import org.openxava.annotations.*;

@MappedSuperclass // Marcada como una superclase mapeada en vez de como una entidad
```

⁶ A partir de OpenXava 4.0 la superclase `Identifiable` está incluida en OpenXava

65 Lección 4: Herencia

```
public class Identifiable {  
  
    @Id @GeneratedValue(generator="system-uuid") @Hidden  
    @GenericGenerator(name="system-uuid", strategy = "uuid")  
    @Column(length=32)  
    private String oid; // La definición de propiedad incluye anotaciones de OX y JPA  
  
    public String getOid() {  
        return oid;  
    }  
  
    public void setOid(String oid) {  
        this.oid = oid;  
    }  
  
}
```

Ahora puedes definir las entidades Author, Category, Detail e Invoice de una manera más sucinta. Para ver un ejemplo tienes el nuevo código para Category en el listado 4.3.

Listado 4.3 Entidad Category refactorizada para extender de Identifiable

```
@Entity  
public class Category extends Identifiable { // Extiende de Identifiable  
    // por tanto no necesita tener una propiedad id  
  
    @Column(length=50)  
    private String description;  
  
    public void setDescription(String description) {  
        this.description = description;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
}
```

La refactorización es extremadamente simple. Category ahora descende de Identifiable y hemos quitado el campo oid y los métodos getOid() y setOid(). De esta forma, no solo tu código es más corto, sino también más elegante, porque estás declarando tu clase como identificable (el qué, no el cómo), y has quitado de tu clase de negocio un código que era un tanto técnico.

Ahora, puedes aplicar esta misma refactorización a las entidades Author, Detail e Invoice. Además, a partir de ahora extenderás la mayoría de tus entidades de la superclase mapeada Identifiable.

Has aprendido, pues, que una superclase mapeada es una clase normal y corriente con anotaciones de mapeo JPA que puedes usar como clase base para tus entidades. También has aprendido como usar una superclase mapeada para

simplificar tu código.

4.2 Herencia de entidades

Una entidad puede heredar de otra entidad. Esta herencia de entidades es una herramienta muy útil para simplificar tu modelo. Vamos a usarla para añadir una nueva entidad, Order, a tu aplicación Invoicing.

4.2.1 Nueva entidad Order

Queremos añadir un nuevo concepto a la aplicación Invoicing: pedido (*order*). Mientras que una factura es algo que quieres cobrar a tu cliente, un pedido es algo que tu cliente te ha solicitado. Estos dos conceptos están fuertemente unidos, porque cobrarás por las cosas que tu cliente te ha pedido, y tú le has servido.

Sería interesante poder tratar pedidos en tu aplicación, y asociar estos pedidos con sus correspondientes facturas. Tal como muestra el diagrama UML de la figura 4.1 y el código Java del listado 4.4.

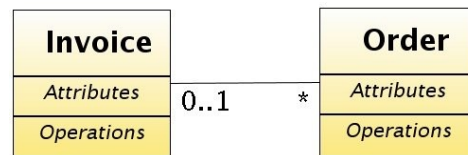


Figura 4.1 UML de Order e Invoice

Listado 4.4 Relación entre Invoice y Order

```

@Entity
public class Invoice {

    @OneToMany(mappedBy="invoice")
    private Collection<Order> orders;
    ...
}

@Entity
public class Order {

    @ManyToOne // Sin inicialización vaga (lazy fetching) (1)
    private Invoice invoice;
}

```

Es decir, una factura tiene varios pedidos, y un pedido puede referenciar a una factura. Fíjate como no usamos inicialización vaga (*lazy fetching*) para la referencia invoice (1), esto es por un bug de Hibernate cuando la referencia contiene la relación bidireccional (es decir, es la referencia declarada en el atributo mappedBy del @OneToMany).

¿Cómo es Order? Bien, tiene un cliente, unas líneas de detalle con producto y cantidad, un año y un número. Algo así como lo que hay en la figura 4.2

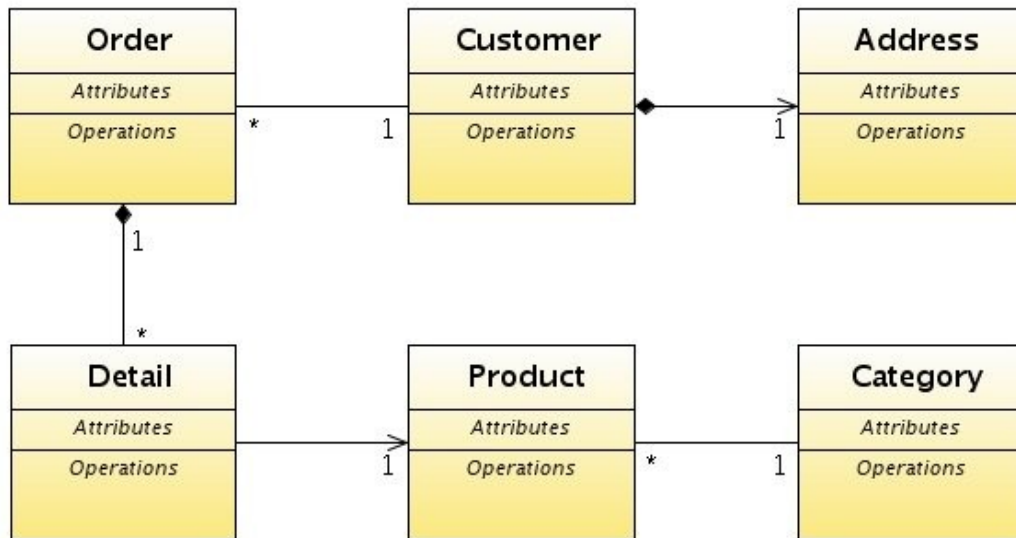


Figura 4.2 Diagrama UML para Order

Curiosamente, este diagrama UML es idéntico al diagrama de Invoice (que puedes ver en la figura 2.1 de la sección 2.1). Es decir, para crear tu entidad Order puedes copiar y pegar la clase Invoice, y asunto zanjado. Pero, ¡espera un momento! ¿“Copiar y pegar” no era un pecado mortal? Hemos de encontrar una forma de reutilizar Invoice para Order.

4.2.2 CommercialDocument como entidad abstracta

Una manera práctica de reutilizar el código de Invoice para Order es usando herencia, además es una excusa perfecta para aprender lo fácil que es usar la herencia con JPA y OpenXava.

En la mayoría de las culturas orientadas a objetos has de observar el precepto sagrado *es un*⁷. Esto significa que no podemos hacer que Invoice herede de Order, porque una Invoice no es un Order, y por la misma regla no podemos hacer que Order descienda de Invoice. La solución para este caso es crear una clase base para ambos, Order y Invoice. Llamaremos a esta clase CommercialDocument.

En la figura 4.3 puedes ver el diagrama UML para CommercialDocument, y en el listado 4.5 tienes

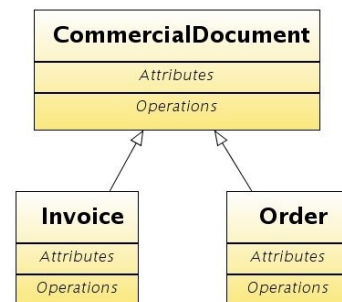


Figura 4.3 UML para CommercialDocument

⁷ La cultura Eiffel no observa la norma *es un*. Es decir *es un* no es una regla absoluta en el universo OO

la misma idea expresada con Java.

Listado 4.5 Usar herencia para definir Order e Invoice

```
public class CommercialDocument { ... }
public class Order extends CommercialDocument { ... }
public class Invoice extends CommercialDocument { ... }
```

Empecemos a refactorizar el código actual. Primero, renombra (usando *Refactor > Rename*) Invoice como CommercialDocument. Después, edita el código de CommercialDocument para declararla como una clase abstracta, tal como muestra el listado 4.6.

Listado 4.6 CommercialDocument es una clase abstracta

```
abstract public class CommercialDocument ← // Añadimos el modificador abstract
```

Queremos crear instancias de Invoice y Order, pero no queremos crear instancias de CommercialDocument directamente, por eso la declaramos abstracta.

4.2.3 Invoice refactorizada para usar herencia

Ahora, has de crear la entidad Invoice extendiéndola de CommercialDocument. Puedes ver el nuevo código de Invoice en el listado 4.7.

Listado 4.7 La entidad Invoice ahora extiende de CommercialDocument

```
@Entity
public class Invoice extends CommercialDocument {
}
}
```

Invoice tiene ahora un código bastante breve, de hecho el cuerpo de la clase está, por ahora, vacío.

Este nuevo código necesita un esquema de base de datos ligeramente diferente, ahora las facturas y los pedidos se almacenarán en la misma tabla (la tabla CommercialDocument) usando una columna discriminador. Por tanto has de borrar las viejas tablas ejecutando las sentencias SQL en el listado 4.8.

Listado 4.8 Sentencia SQL para borrar las viejas tablas de Invoice

```
drop table detail;
drop table invoice;
```

Puedes ejecutar estas sentencias SQL desde la perspectiva de Eclipse o desde tu herramienta de administración de base de datos favorita.

Después de borrar las tablas viejas, ejecuta la tarea ant updateSchema para regenerar todas las tablas necesarias. Ya puedes ejecutar el módulo Invoice y

verlo funcionando en tu navegador. Lanza también InvoiceTest. Tiene que salirte verde.

4.2.4 Crear Order usando herencia

Gracias a CommercialDocument el código para Order es más sencillo que el mecanismo de un sonajero. Míralo en el listado 4.9.

Listado 4.9 Entidad Order que extiende de CommercialDocument

```
@Entity
public class Order extends CommercialDocument {

}
```

Después de escribir la clase Order del listado 4.9, aunque de momento esté vacía, ya puedes usar el módulo Order desde tu navegador. Sí, a partir de ahora crear una nueva entidad con una estructura parecida a Invoice, es decir cualquier entidad para un documento comercial, es simple y rápido. Un buen uso de la herencia es una forma elegante de tener un código más simple.

El módulo Order funciona perfectamente, pero tiene un pequeño problema. El nuevo número de pedido lo calcula a partir del último número de factura, no de pedido. Esto es así porque el calculador para el siguiente número lee de la entidad Invoice. Una solución obvia es mover la definición de la propiedad number de CommercialDocument a Invoice y Order. Aunque, no lo vamos a hacer así, porque en la lección 5 refinaremos la forma de calcular el número de documento, de momento simplemente haremos un pequeño ajuste en el código actual para que no falle. Edita la clase NextNumberForYearCalculator y en la consulta cambia “Invoice” por “CommercialDocument”, dejando el método calculate() como en el listado 4.10.

Listado 4.10 NextNumberYearCalculator.calculate() con CommercialDocument

```
public Object calculate() throws Exception {
    Query query = XPersistence.getManager()
        .createQuery(
            "select max(i.number) from " +
            "CommercialDocument i " + // CommercialDocument en vez de Invoice
            "where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    return lastNumber == null ? 1 : lastNumber + 1;
}
```

Ahora buscamos el número máximo de cualquier tipo de documento comercial del año para calcular el nuevo número, por lo tanto la numeración es compartida para todos los tipos de documentos. Esto lo mejoraremos en la lección 5 para separar la numeración para facturas y pedidos; y para tener un mejor soporte de

entornos multiusuario.

4.2.5 Convención de nombres y herencia

Fíjate que no has necesitado cambiar el nombre de ninguna propiedad de Invoice para hacer la refactorización. Esto es por el siguiente principio práctico: *No uses el nombre de clase en los nombres de miembro*, por ejemplo, dentro de una clase Account no uses la palabra “Account” en ningún método o propiedad. Mira el listado 4.11.

Listado 4.11 No uses el nombre de la clase en los nombres de miembros

```
public class Account { // No usaremos Account en los nombres de los miembros

    private int accountNumber; // Mal, porque usa "account"
    private int number; // Bien, no usa "account"

    public void cancelAccount() { } ← // Mal, porque usa "Account"
    public void cancel() { } // Bien, no usa "account"
    ...
}
```

Usando esta nomenclatura puedes refactorizar la clase Account en una jerarquía sin renombrar sus miembros, y además puedes escribir código polimórfico con Account.

4.2.6 Asociar Order con Invoice

Hasta ahora, Order e Invoice son exactamente iguales. Vamos a hacerles sus primeras extensiones, que va a ser asociar Order con Invoice, como muestra la figura 4.4. Solo necesitas añadir una referencia desde Order a Invoice. Tienes el código para esto en listado 4.12.

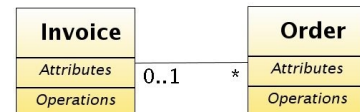


Figura 4.4 Asociación entre Invoice y Order

Listado 4.12 Código íntegro de Order con una referencia a Invoice

```
@Entity
public class Order extends CommercialDocument {

    @ManyToOne
    private Invoice invoice; // Añadida referencia a Invoice

    public Invoice getInvoice() {
        return invoice;
    }

    public void setInvoice(Invoice invoice) {
        this.invoice = invoice;
    }
}
```



```
}
```

Por otra parte en Invoice añadimos una colección de entidades Order. Lo puedes ver en el listado 4.13.

Listado 4.13 Código íntegro de Invoice con una colección de entidades Order

```
@Entity
public class Invoice extends CommercialDocument {

    @OneToMany(mappedBy="invoice")
    private Collection<Order> orders; // Añadimos colección de entidades Order

    public Collection<Order> getOrders() {
        return orders;
    }

    public void setOrders(Collection<Order> orders) {
        this.orders = orders;
    }

}
```

Después de escribir este código tan simple, ya puedes probar estas, recién creadas, relaciones. Primero, actualiza el esquema de la base de datos, y después abre tu navegador y explora los módulos Order e Invoice. Fíjate como al final de la interfaz de usuario de Order tienes una referencia a Invoice. El usuario puede usar esta referencia para asociar una factura al pedido actual. Por otra parte, si exploras el módulo Invoice, verás una colección de pedidos al final. El usuario puede usarla para añadir pedidos a la factura actual.

Intenta añadir pedidos a la factura, y asociar una factura a un pedido. Funciona, aunque la interfaz de usuario es un poco fea, de momento.

4.3 Herencia de vistas

La herencia no solo sirve para reutilizar código Java y mapeos, sino también para reutilizar la definición de la interfaz de usuario, las definiciones @View. Esta sección muestra como funciona la herencia de vistas.

4.3.1 El atributo extendsView

Tanto Order como Invoice usan una interfaz de usuario generada por defecto con todos sus miembros uno por cada línea. Nota como la @View que hemos declarado en CommercialDocument no se ha heredado. Es decir, si no defines una vista para la entidad se genera una por defecto, la @View de la entidad padre no se usa. Tal como muestra el listado 4.14.

Listado 4.14 Las vistas no se heredan por defecto

```
@View(members = "a, b, c;") // Esta vista se usa para visualizar Parent, pero no para Child
public class Parent { ... }

public class Child extends Parent { ... } // Child se visualiza usando la vista
// generada automáticamente, no la vista de Parent
```

Generalmente la vista de la entidad padre “tal cual” no es demasiado útil porque no contiene todas las propiedades de la entidad actual. Por tanto este comportamiento suele venir bien como comportamiento por defecto.

Aunque, en una entidad no trivial necesitas refinar la interfaz de usuario y quizás sea útil heredar (en lugar de copiar y pegar) la vista del padre. Puedes hacer esto usando el atributo `extendsView` en `@View`. Mira el listado 4.15.

Listado 4.15 Usar herencia de vista por medio de `extendsView`

```
@View(members = "a, b, c;") // Esta vista sin nombre es la vista DEFAULT
public class Parent { ... }

@Views({
    @View(name="A" members = "d", // Añade d a la vista heredada
        extendsView = "super.DEFAULT"), // Extienda la vista por defecto de Parent
    @View(name="B" members = "a, b, c; d") // La vista B es igual a la vista A
})
public class Child extends Parent { ... } // Child se visualiza usando la vista
// generada automáticamente, no la vista de Parent
```

Usando `extendsView` los miembros a mostrar serán aquellos en la vista que extendemos más aquellos en `members` de la actual.

Vamos a usar esta característica para definir las vistas para `CommercialDocument`, `Order` e `Invoice`.

4.3.2 Vista para `Invoice` usando herencia

Dado que la `@View` de `CommercialDocument` no se hereda, la interfaz de usuario actual para `Invoice` es bastante fea: todos los miembros, uno por línea. Vamos a definir una interfaz de usuario mejor. Una vista parecida a la de la lección 5, pero añadiendo una pestaña para pedidos. Queremos algo como lo que muestra la figura 4.5.

73 Lección 4: Herencia

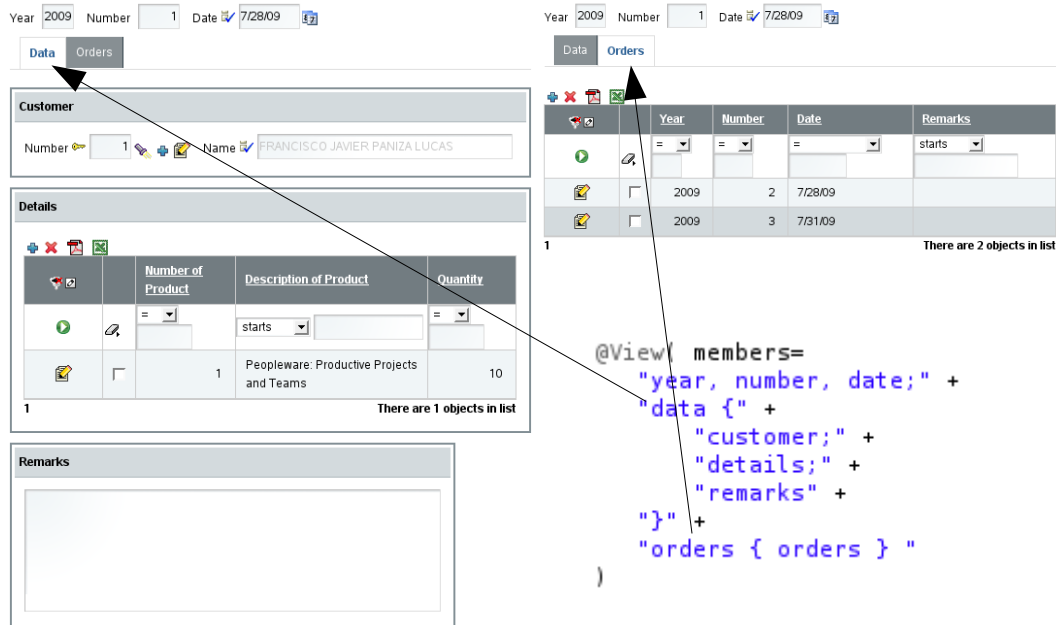


Figura 4.5 Interfaz de usuario para Invoice con orders

Nota que ponemos todos los miembros de la parte de CommercialDocument de Invoice en la cabecera y la primera pestaña (*data*), y la colección de pedidos en la otra pestaña. El listado 4.16 muestra la forma de definir esto sin herencia.

Listado 4.16 Vista para Invoice sin herencia de vistas

```
@View( members=
    "year, number, date;" +
    "data {" +
        "customer;" +
        "details;" +
        "remarks" +
    "}" +
    "orders { orders } "
)
public class Invoice extends CommercialDocument {
```

Puedes notar como todos los miembros, excepto la parte de orders, son comunes para todos los CommercialDocument. Por lo tanto, vamos a mover esta parte común a CommercialDocument y redefinir la vista usando herencia.

Quita el viejo @View de CommercialDocument, y escribe el que hay en el listado 4.17.

Listado 4.17 Vista para CommercialDocument

```
@View(members=
    "year, number, date," + // Los miembros para la cabecera en una línea
    "data {" + // Una pestaña 'data' para los datos principales del documento
```

```

        "customer;" +
        "details;" +
        "remarks" +
    }"
)
abstract public class CommercialDocument extends Identifiable {

```

Esta vista indica como distribuir los datos comunes para todos los documentos comerciales. Ahora podemos redefinir la vista de Invoice a partir de esta. Mira el listado 4.18.

Listado 4.18 Definición de la vista de Invoice usando herencia de vistas

```

@View(extendsView="super.DEFAULT", // Extiende de la vista de CommercialDocument
      members="orders { orders }" // Añadimos orders dentro de una pestaña
)
public class Invoice extends CommercialDocument {

```

De esta forma declarar la vista para Invoice es más corto, es más, la distribución común para Order, Invoice y todos los demás posibles CommercialDocument está en un único sitio, por tanto, si añades una nueva propiedad a CommercialDocument solo has de tocar la vista de CommercialDocument.

4.3.3 Vista para Order usando herencia

Ahora que tienes una vista adecuada para CommercialDocument, declarar una vista para Order es lo más fácil del mundo. La figura 4.6 muestra la vista que

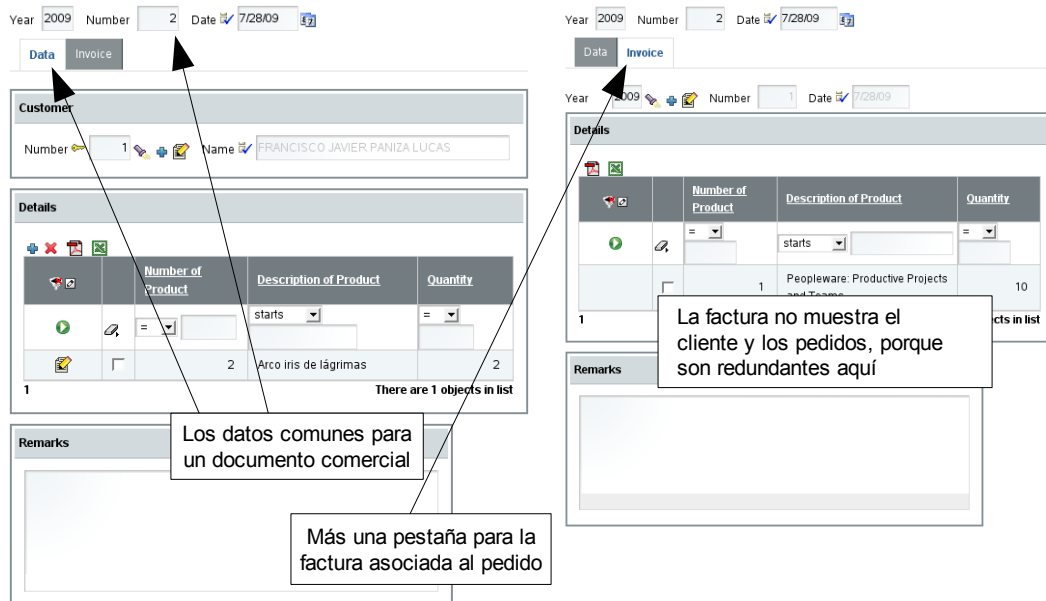


Figura 4.6 Interfaz de usuario para Order con una referencia a Invoice

75 Lección 4: Herencia

queremos. Una vista con toda la información del pedido, y su factura asociada en otra pestaña.

Para obtener este resultado, puedes definir la vista de Order extendiendo la vista por defecto de CommercialDocument, y añadiendo la referencia a factura, tal como muestra el listado 4.19.

Listado 4.19 Definición de vista de Order usando herencia de vistas

```
@View(extendsView="super.DEFAULT", // Extiende de la vista de CommercialDocument
      members="invoice { invoice } " // Añadimos invoice dentro de una pestaña
)
public class Order extends CommercialDocument {
```

Con esto conseguimos toda la información de CommercialDocument más una pestaña con la factura.

4.3.4 Usar @ReferenceView y @CollectionView para refinar vistas

Queremos que cuando un pedido sea editado desde la interfaz de usuario de Invoice la vista a usar sea simple, sin cliente ni factura, porque estos datos son redundantes en este caso. Mira la figura 4.7.



Figura 4.7 Vista para editar Order desde Invoice tiene que ser más simple

Para obtener este resultado define una vista más simple en Order (listado 4.20), y referénciala desde la colección orders en Invoice (listado 4.21).

Listado 4.20 Vista NoCustomerNoInvoice en Order para ser usada desde Invoice

```
@Views({ // Para declarar más de una vista
  @View( extendsView="super.DEFAULT", // La vista por defecto
        members="invoice { invoice } "
  ),
  @View( name="NoCustomerNoInvoice", // Una vista llamada NoCustomerNoInvoice
```

```

        members=
            "year, number, date;" +
            "details;" +
            "remarks"
    )
})
public class Order extends CommercialDocument {

```

Esta nueva vista definida en Order llamada NoCustomerNoInvoice puede ser referenciada desde Invoice para visualizar o editar elementos individuales de la colección orders usando @CollectionView. Míralo en el listado 4.21.

Listado 4.21 Usar la vista NoCustomerNoInvoice de Order desde Invoice

```

@OneToMany(mappedBy="invoice")
@CollectionView("NoCustomerNoInvoice") // Esta vista se usa para visualizar orders
private Collection<Order> orders;

```

Y tan solo con este código la colección orders usará una vista más apropiada de Invoice para editar elementos individuales (figura 4.7).

Además, queremos que desde la interfaz de usuario de Order la factura no muestre el cliente y los pedidos, porque son datos redundantes en este caso. Para conseguirlo, vamos a definir una vista más simple en Invoice (listado 4.22), y referenciarla desde la referencia invoice en Order (listado 4.23).

Listado 4.22 Nueva vista NoCustomerNoOrders en Invoice para usarse en Order

```

@Views({ // Para declarar más de una vista
    @View( extendsView="super.DEFAULT", // La vista por defecto
        members="orders { orders } "
    ),
    @View( name="NoCustomerNoOrders", // Una vista llamada NoCustomerNoOrders
        members= // que no incluye customer y orders
            "year, number, date;" + // Ideal para usarse desde Order
            "details;" +
            "remarks"
        )
    })
public class Invoice extends CommercialDocument {

```

Esta nueva vista definida en Invoice llamada NoCustomerNoOrders puede ser referenciada desde Order para visualizar la referencia Invoice usando @ReferenceView. Lo puedes ver en el listado 4.23.

Listado 4.23 Usar la vista NoCustomerNoOrders de Invoice desde Order

```

public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders") // Esta vista se usa para visualizar invoice
    private Invoice invoice;

    ...

```

```
}
```

Ahora la referencia `invoice` será visualizada apropiadamente desde `Order` y así tendrás la interfaz de usuario deseada.

4.4 Herencia en las pruebas JUnit

`Invoice` ha sido refactorizada para usar herencia, y también hemos usado herencia para añadir una nueva entidad, `Order`. Además, esta entidad `Order` tiene relación con `Invoice`. Lo cual es una nueva funcionalidad, por ende has de probar todas estas nuevas características.

Dado que `Invoice` y `Order` tienen bastantes cosas en común (la parte de `CommercialDocument`) podemos refactorizar las pruebas para usar herencia, y así eludir el dañino “copiar y pegar” también en tu código de prueba.

4.4.1 Crear una prueba de módulo abstracta

Si examinas la prueba para crear una factura, en el método `testCreate()` de `InvoiceTest` (sección 3.6). Puedes notar que probar la creación de una factura es exactamente igual que probar la creación de un pedido. Porque, ambos tienen año, número, fecha, cliente, detalles y observaciones. Por tanto, aquí la herencia es una buena herramienta para la reutilización de código.

Vamos a renombrar `InvoiceTest` como `CommercialDocumentTest`, y entonces crearemos `InvoiceTest` y `OrderTest` a partir de él. Puedes ver el diagrama UML de esta idea en la figura 4.8.

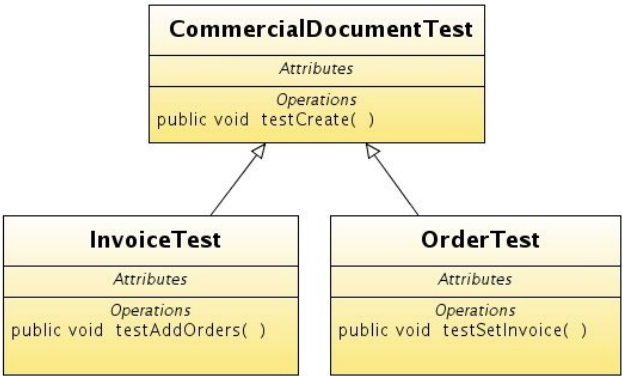


Figura 4.8 Jerarquía de `CommercialDocumentTest`

Primero renombra tu actual clase `InvoiceTest` a `CommercialDocumentTest`, y después haz los cambios indicados en el listado 4.24.

Listado 4.24 `CommercialDocumentTest` creado renombrando el viejo `InvoiceTest`

```
abstract public class CommercialDocumentTest // Añade abstract a la definición de clase
    extends ModuleTestBase {
```

```

private String number;

public CommercialDocumentTest(
    String testName,
    String moduleName) // moduleName añadido como argumento en el constructor
{
    super(testName, "Invoicing", moduleName); // Envía el moduleName
}

public void testCreate() throws Exception { ... } // Como el original

private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0);
    execute("Collection.new",
        "viewObject=xava_view_section0_details"); // Cambia xava_view_details
                                                    // por xava_view_section0_details

    // El resto del método como el original
    ...
}

private String getNumber() {
    if (number == null) {
        Query query = getManager().
            createQuery(
                "select max(i.number) from " +
                "CommercialDocument i " + // Invoice cambiada por CommercialDocument
                "i where i.year = :year");
        query.setParameter("year",
            Dates.getYear(new Date()));
        Integer lastNumber = (Integer)
            query.getSingleResult();
        if (lastNumber == null) lastNumber = 0;
        number = Integer.toString(lastNumber + 1);
    }
    return number;
}

private void remove() throws Exception { ... } // Como el original

private void verifyCreated() throws Exception { ... } // Como el original

private void save() throws Exception { ... } // Como el original

private void setOtherProperties()
    throws Exception { ... } // Como el original

private void chooseCustomer() throws Exception { ... } // Como el original

private void verifyDefaultValues()
    throws Exception { ... } // Como el original

private String getCurrentYear() { ... } // Como el original

private String getCurrentDate() { ... } // Como el original
}

```

Como ves en el listado 4.24 has tenido que hacer unos pocos cambios para

79 Lección 4: Herencia

adaptar `CommercialDocumentTest`. Primero, la has declarado abstracta, de esta forma esta clase no es ejecutada por Eclipse como una prueba JUnit, es solo válida como clase base para crear pruebas, pero ella misma no es una prueba.

Otro cambio importante nos lo encontramos en el constructor, donde ahora tienes `moduleName` en vez de “Invoice”, así puedes usar esta prueba para `Order`, `Invoice` o cualquier otro módulo que quieras. Los otros cambios son simples detalles: has de usar `xava_view_section0_details` para `viewObject` cuando llamas a las acciones de la colección, porque ahora la colección `details` está en la primera pestaña (`section0`), y has de cambiar “Invoice” por “ComercialDocument” en la consulta para obtener el siguiente número.

Ahora ya tienes una clase base lista para crear los módulos de prueba para `Order` e `Invoice`. Hagámoslo sin más dilación.

4.4.2 Prueba base abstracta para crear pruebas de módulo concretas

Crear tu primera versión para `OrderTest` e `InvoiceTest` es simplemente extender de `CommercialDocumentTest`. Nada más. Mira `InvoiceTest` en el listado 4.25.

Listado 4.25 InvoiceTest usando herencia

```
public class InvoiceTest extends CommercialDocumentTest {  
    public InvoiceTest(String testName) {  
        super(testName, "Invoice");  
    }  
}
```

Y `OrderTest` en el listado 4.26.

Listado 4.26 OrderTest usando herencia

```
public class OrderTest extends CommercialDocumentTest {  
    public OrderTest(String testName) {  
        super(testName, "Order");  
    }  
}
```

Ejecuta estas dos prueba y verás como `testCreate()`, heredado de `CommercialDocumentTest`, se ejecuta en ambos casos, contra su módulo correspondiente. Con esto estamos probando el comportamiento común para `Order` e `Invoice`. Probemos ahora la funcionalidad particular de cada uno.

4.4.3 Añadir nuevas pruebas a las pruebas de módulo extendidas

Hasta ahora hemos probado como crear una factura y un pedido. En esta sección probaremos como añadir pedidos a una factura en el módulo Invoice, y como establecer la factura a un pedido en el módulo Order.

Para probar como añadir un pedido a una factura añade el método `testAddOrdersMethod()` del listado 4.27 a `InvoiceTest`.

Listado 4.27 Probar añadir pedidos a una factura en InvoiceTest

```
public void testAddOrders() throws Exception {
    assertListNotEmpty(); // Esta prueba confía en que ya existen facturas
    execute("List.orderBy", "property=number"); // Para usar siempre el mismo pedido
    execute("Mode.detailAndFirst"); // Va al modo detalle editando la primera factura
    execute("Sections.change", "activeSection=1"); // Cambia a la pestaña 1
    assertCollectionRowCount("orders", 0); // Esta factura no tiene pedidos
    execute("Collection.add", // Pulsa el botón para añadir un nuevo pedido, esto te lleva
           "viewObject=xava_view_section1_orders"); // a la lista de pedidos
    execute("AddToCollection.add", "row=0"); // Escoge el primer pedido de la lista
    assertCollectionRowCount("orders", 1); // El pedido se ha añadido a la factura
    checkRowCollection("orders", 0); // Marca el pedido, para borrarlo
    execute("Collection.removeSelected", // Borra el pedido recién añadido
           "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0); // El pedido ha sido borrado
}
```

En este caso asumimos que hay al menos una factura, y que la primera factura de la lista no tiene pedidos. Antes de ejecutar esta prueba, si no tienes facturas todavía, crea una sin pedidos, o si ya tienes facturas, asegúrate de que la primera no tiene pedidos.

Para probar como asignar una factura a un pedido añade el método `testSetInvoice()` del listado 4.28 a `OrderTest`.

Listado 4.28 Probar asignar una factura a un pedido en OrderTest

```
public void testSetInvoice() throws Exception {
    assertListNotEmpty(); // Esta prueba confía en que existen pedidos
    execute("Mode.detailAndFirst"); // Va a modo detalle editando la primera factura
    execute("Sections.change", "activeSection=1"); // Cambia a la pestaña 1
    assertValue("invoice.number", ""); // Este pedido todavía no tiene
    assertValue("invoice.year", ""); // una factura asignada
    execute("Reference.search", // Pulsa en el botón para buscar la factura, esto te
           "keyProperty=invoice.year"); // lleva a la lista de facturas
    String year = getValueInList(0, "year"); // Memoriza el año y el número de
    String number = getValueInList(0, "number"); // la primera factura de la lista
    execute("ReferenceSearch.choose", "row=0"); // Escoge la primera factura
    assertValue("invoice.year", year); // Al volver al detalle del pedido verificamos
    assertValue("invoice.number", number); // que la factura ha sido seleccionada
}
```

81 Lección 4: Herencia

En este caso asumimos que hay al menos un pedido, y el primer pedido de la lista no tiene factura. Antes de ejecutar esta prueba, si no tienes pedidos, crea uno sin factura, o si ya tienes pedidos, asegúrate de que el primero no tiene factura.

Con esto ya tienes tus pruebas listas. Ejecútalas, y obtendrás el resultado de la figura 4.9. Fíjate que la prueba base `CommercialDocumentTest` no se muestra porque es abstracta. Y `testCreate()` de `CommercialDocumentTest` se ejecuta para `InvoiceTest` y `OrderTest`.

No solo has adaptado tu código de pruebas al nuevo código de Invoicing, sino que también has aprendido como usar herencia en el mismo código de pruebas.

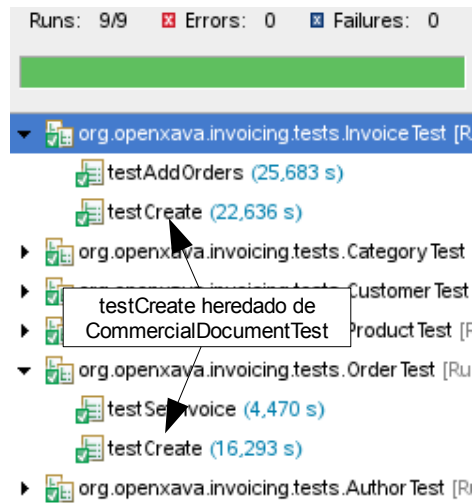


Figura 4.9 Pruebas de Invoicing

que también has aprendido como usar

4.5 Resumen

Esta lección te ha mostrado algunos ejemplos prácticos sobre como usar herencia con Java y JPA para simplificar tu código. Hemos usado la configuración por defecto de JPA para la herencia, aunque puedes refinar el comportamiento de JPA para la herencia con anotaciones como `@Inheritance`, `@DiscriminatorColumn`, `@DiscriminatorValue`, etc. Para aprender más acerca de la herencia en JPA puedes leer la documentación del apéndice B.

OpenXava, al menos en la versión 3.1.x, solo soporta la estrategia de tabla única de JPA, este quiere decir que todos los datos de una jerarquía se almacenan en la misma tabla.